Algorithms For Interviews

A Problem Solving Approach

Adnan Aziz
Amit Prakash

algorithmsforinterviews.com

Adnan Aziz is a professor at the Department of Electrical and Computer Engineering at The University of Texas at Austin, where he conducts research and teaches classes in applied algorithms. He received his PhD from The University of California at Berkeley; his undergraduate degree is from IIT Kanpur. He has worked at Google, Qualcomm, IBM, and several software startups. When not designing algorithms, he plays with his children, Laila, Imran, and Omar.

Amit Prakash is a Member of the Technical Staff at Google, where he works primarily on machine learning problems that arise in the context of online advertising. Prior to that he worked at Microsoft in the web search team. He received his PhD from The University of Texas at Austin; his undergraduate degree is from IIT Kanpur. When he is not improving the quality of ads, he indulges in his passions for puzzles, movies, travel, and adventures with his wife.

This book was typeset by the authors using Lesley Lamport's LaTeXdocument preparation system and Peter Wilson's Memoir class. The cover design was done using Inkscape. MacOSaiX was used to create the front cover image; it approximates Shela Nye's portrait of Alan Turing using a collection of public domain images of famous computer scientists and mathematicians. The graphic on the back cover was created by Nidhi Rohatgi.

The companion website for the book includes a list of known errors for each version of the book. If you come across a technical error, please write to us and we will cheerfully send you $0.42. Please refer to the website for details.

Version 1.0.0 (September 21, 2010)

Website: `http://algorithmsforinterviews.com`

`ISBN:978-0-557-63040-0`

Table of Contents

Prologue

Let's begin with the picture on the front cover. You may have observed that the portrait of Alan Turing is constructed from a number of pictures ("tiles") of great computer scientists and mathematicians.

Suppose you were asked in an interview to design a program that takes an image and a collection of $s \times s$-sized tiles and produce a mosaic from the tiles that resembles the image. A good way to begin may be to partition the image into $s \times s$-sized squares, compute the average color of each such image square, and then find the tile that is closest to it in the color space. Here distance in color space can be $L2$-norm over Red-Green-Blue (RGB) intensities for the color. As you look more carefully at the problem, you might conclude that it would be better to match each tile with an image square that has a similar structure. One way could be to perform a coarse pixelization (2×2 or 3×3) of each image square and finding the tile that is "closest" to the image square under a distance

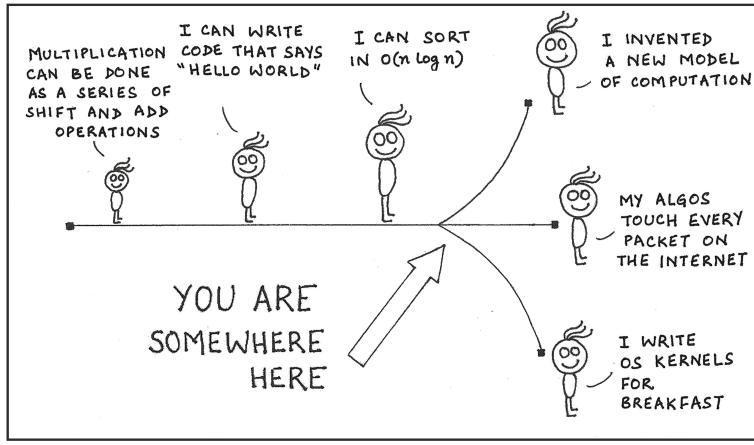

Figure 1. Evolution of a computer scientist

function defined over all pixel colors (for example, $L2$-norm over RGB values for each pixel). Depending on how you represent the tiles, you end up with the problem of finding the closest point from a set of points in a k-dimensional space.

If there are m tiles and the image is partitioned into n squares, then a brute-force approach would have $O(m \cdot n)$ time complexity. You could improve on this by first indexing the tiles using an appropriate search tree. A more detailed discussion on this approach is presented in Problem 8.1 and its solution.

If in a 45-60 minute interview, you can work through the above ideas, write some pseudocode for your algorithm, and analyze its complexity, you would have had a fairly successful interview. In particular, you would have demonstrated to your interviewer that you possess several key skills:

- The ability to rigorously formulate real-world problems.
- The skills to solve problems and design algorithms.
- The tools to go from an algorithm to a working program.
- The analytical techniques required to determine the computational complexity of your solution.

Book Overview

Algorithms for Interviews (AFI) aims to help engineers interviewing for software development positions. The primary focus of AFI is algorithm design. The entire book is presented through problems interspersed with discussions. The problems cover key concepts and are well-motivated, challenging, and fun to solve.

We do not emphasize platforms and programming languages since they differ across jobs, and can be acquired fairly easily. Interviews at most large software companies focus more on algorithms, problem solving, and design skills than on specific domain knowledge. Also, platforms and programming languages can change quickly as requirements change but the qualities mentioned above will always be fundamental to any successful software endeavor.

The questions we present should all be solvable within a one hour interview and in many cases, take substantially less time. A question may take more or less time to complete, depending on the amount of coding that is asked for.

Our solutions vary in terms of detail—for some problems we present detailed implementations in Java/C++/Python; for others, we simply sketch solutions. Some use fairly technical machinery, e.g., max-flow, randomized analysis, etc. You will encounter such problems only if you

claim specialized knowledge, e.g., graph algorithms, complexity theory, etc.

Interviewing is about more than being able to design algorithms quickly. You also need to know how to present yourself, how to ask for help when you are stuck, how to come across as being excited about the company, and knowing what you can do for them. We discuss the non-technical aspects of interviewing in Chapter 12. You can practice with friends or by yourself; in either case, be sure to time yourself. Interview at as many places as you can without it taking away from your job or classes. The experience will help you and you may discover you like companies that you did not know much about.

Although an interviewer may occasionally ask a question directly from AFI, you should not base your preparation on memorizing solutions from AFI. We sincerely hope that reading this book will be enjoyable and improve your algorithm design skills. The end goal is to make you a better engineer as well as better prepared for software interviews.

Level and Prerequisites

Most of AFI requires its readers to have basic familiarity with algorithms taught in a typical undergraduate-level algorithms class. The chapters on meta-algorithms, graphs, and intractability use more advanced machinery and may require additional review.

Each chapter begins with a review of key concepts. This review is not meant to be comprehensive and if you are not familiar with the material, you should first study the corresponding chapter in an algorithms textbook. There are dozens of such texts and our preference is to master one or two good books rather than superficially sample many. We like *Algorithms* by Dasgupta, Papadimitriou, and Vazirani because it is succinct and beautifully written; *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein is more detailed and serves as a good reference.

Since our focus is on problems that can be solved in an interview relatively completely, there are many elegant algorithm design problems which we do not include. Similarly, we do not have any straightforward review-type problems; you may want to brush up on these using introductory programming and data-structures texts.

The field of algorithms is vast and there are many specialized topics, such as computational geometry, numerical analysis, logic algorithms, etc. Unless you claim knowledge of such topics, it is highly unlikely that you will be asked a question which requires esoteric knowledge. While an interview problem may seem specialized at first glance, it is invariably the case that the basic algorithms described in this book are sufficient to solve it.

Acknowledgments

The problems in this book come from diverse sources—our own experiences, colleagues, friends, papers, books, Internet bulletin boards, etc. To paraphrase Paul Halmos from his wonderful book *Problems for Mathematicians, Young and Old*: "I do not give credits—who discovered what? Who was first? Whose solution is the best? It would not be fair to give credit in some cases and not in others. No one knows who discovered the theorem that bears Pythagoras' name and it does not matter. The beauty of the subject speaks for itself and so be it."

One person whose help and support has improved the quality of this book and made it fun to read is our cartoonist, editor, and proofreader, Nidhi Rohatgi. Several of our friends and students gave feedback on this book—we would especially like to thank Ian Varley, who wrote solutions to several problems, and Senthil Chellappan, Gayatri Ramachandran, and Alper Sen for proofreading several chapters.

We both want to thank all the people who have been a source of enlightenment and inspiration to us through the years.

I, Adnan Aziz, would like to thank teachers, friends, and students from IIT Kanpur, UC Berkeley, and UT Austin. I would especially like to thank my friends Vineet Gupta and Vigyan Singhal, and my teachers Robert Solovay, Robert Brayton, Richard Karp, Raimund Seidel, and Somenath Biswas for introducing me to the joys of algorithms. My co-author, Amit Prakash, has been a wonderful collaborator—this book is a testament to his intellect, creativity, and enthusiasm.

I, Amit Prakash, have my co-author and mentor, Adnan Aziz, to thank the most for this book. To a great extent, my problem solving skills have been shaped by Adnan. There have been occasions in life when I would not have made through without his help. He is also the best possible collaborator I can think of for any intellectual endeavor.

Over the years, I have been fortunate to have great teachers at IIT Kanpur and UT Austin. I would especially like to thank Professors Scott Nettles, Vijaya Ramachandran, and Gustavo de Veciana. I would also like to thank my friends and colleagues at Google, Microsoft, and UT Austin for all the stimulating conversations and problem solving sessions. Lastly and most importantly, I want to thank my family who have been a constant source of support, excitement, and joy for all my life and especially during the process of writing this book.

ADNAN AZIZ
adnan@algorithmsforinterviews.com

AMIT PRAKASH
amit@algorithmsforinterviews.com

Problem Solving Techniques

> It's not that I'm so smart, it's just that I stay with problems longer.
>
> A. Einstein.

Developing problem solving skills is like learning to play a musical instrument—a book or a teacher can point you in the right direction, but only your hard work will take you where you want to go. Like a musician, you need to know underlying concepts but theory is no substitute for practice; for this reason, AFI consists primarily of problems.

Great problem solvers have skills that cannot be captured by a set of rules. Still, when faced with a challenging algorithm design problem it is helpful to have a small set of general principles that may be applicable. We enumerate a collection of such principles in Table 1. Often, you may have to use more than one of these techniques.

We will now look at some concrete examples of how these techniques can be applied.

DIVIDE-AND-CONQUER AND GENERALIZATION

A triomino is formed by joining three unit-sized squares in an L-shape. A mutilated chessboard (henceforth 8×8 Mboard) is made up of 64 unit-sized squares arranged in an 8×8 square, minus the top left square. Supposed you are asked to design an algorithm which computes a placement of 21 triominos that covers the 8×8 Mboard. (Since there are 63 squares in the 8×8 Mboard and we have 21 triominos, a valid placement cannot have overlapping triominos or triominos which extend out of the 8×8 Mboard.)

Divide-and-conquer is a good strategy to attack this problem. Instead of the 8×8 Mboard, let's consider an $n \times n$ Mboard. A 2×2 Mboard can be covered with 1 triomino since it is of the same exact shape. You may hypothesize that a triomino placement for an $n \times n$ Mboard with the top left square missing can be used to compute a placement for an $n+1 \times n+1$

Technique	Description
Divide-and-conquer	Can you divide the problem into two or more smaller independent subproblems and solve the original problem using solutions to the subproblems?
Recursion, dynamic programming	If you have access to solutions for smaller instances of a given problem, can you easily construct a solution to the problem?
Case analysis	Can you split the input/execution into a number of cases and solve each case in isolation?
Generalization	Is there a problem that subsumes your problem and is easier to solve?
Data-structures	Is there a data-structure that directly maps to the given problem?
Iterative refinement	Most problems can be solved using a brute-force approach. Can you formalize such a solution and improve upon it?
Small examples	Can you find a solution to small concrete instances of the problem and then build a solution that can be generalized to arbitrary instances?
Reduction	Can you use a problem with a known solution as a subroutine?
Graph modeling	Can you describe your problem using a graph and solve it using an existing algorithm?
Write an equation	Can you express relationships in your problem in the form of equations (or inequalities)?
Auxiliary elements	Can you add some new element to your problem to get closer to a solution?
Variation	Can you solve a slightly different problem and map its solution to your problem?
Parallelism	Can you decompose your problem into subproblems that can be solved independently on different machines?
Caching	Can you store some of your computation and look it up later to save work?
Symmetry	Is there symmetry in the input space or solution space that can be exploited?

Table 1. Common problem solving techniques.

Mboard. However you will quickly see that this line of reasoning does not lead you anywhere.

Another hypothesis is that if a placement exists for an $n \times n$ Mboard, then one also exists for a $2n \times 2n$ Mboard. This does work: take 4 $n \times n$ Mboards and arrange them to form a $2n \times 2n$ square in such a way that three of the Mboards have their missing square set towards the center and one Mboard has its missing square outward to coincide with the missing corner of a $2n \times 2n$ Mboard. The gap in the center can be covered with a triomino and, by hypothesis, we can cover the 4 $n \times n$ Mboards with triominos as well. Hence a placement exists for any n that is a power of 2. In particular, a placement exists for the $2^3 \times 2^3$ Mboard; the recursion used in the proof can be directly coded to find the actual coverings as well. Observe that this problem demonstrates divide-and-conquer as well as generalization (from 8×8 to $2^n \times 2^n$).

Recursion and Dynamic Programming

Suppose you were to design an algorithm that takes an unparenthesized expression containing addition and multiplication operators, and returns the parenthesization that maximizes the value of the expression. For example, the expression $5 - 3 \cdot 4 + 6$ yields any of the following values:

$$\begin{aligned} -25 &= 5 - \big(3 \cdot (4 + 6)\big) \\ -13 &= 5 - \big((3 \cdot 4) + 6\big) \\ 20 &= (5 - 3) \cdot (4 + 6) \\ -1 &= \big(5 - (3 \cdot 4)\big) + 6 \\ 14 &= \big((5 - 3) \cdot 4\big) + 6 \end{aligned}$$

If we recursively compute the parenthesization for each subexpression that maximizes its value, it is easy to identify the optimum top level parenthesization—parenthesize on each side of the operators and determine which operator maximizes the value of the total expression.

Recursive computation of the maximizing parenthesization for subexpressions leads to repeated calls with identical arguments. Dynamic programming avoids these repeated computations; refer to Problem 3.11 for a detailed exposition.

Case Analysis

You are given a set S of 25 distinct integers and a CPU that has a special instruction, `SORT5`, that can sort 5 integers in one cycle. Your task is to identify the 3 largest integers in S using `SORT5` to compare and sort subsets of S; furthermore, you must minimize the number of calls to `SORT5`.

If all we had to compute was the largest integer in the set, the optimum approach would be to form 5 disjoint subsets $S_1, \ldots, S_5$ of S, sort each subset, and then sort $\{\max S_1, \ldots, \max S_5\}$. This takes 6 calls to `SORT5` but leaves ambiguity about the second and third largest integers.

It may seem like many calls to `SORT5` are still needed. However if you do a careful case analysis and eliminate all $x \in S$ for which there are at least 3 integers in S larger than x, only 5 integers remain and hence just one more call to `SORT5` is needed to compute the result. Details are given in the solution to Problem 2.5.

FIND A GOOD DATA STRUCTURE

Suppose you are given a set of files, each containing stock quote information. Each line contains starts with a timestamp. The files are individually sorted by this value. You are to design an algorithm that combines these quotes into a single file R containing these quotes, sorted by the timestamps.

This problem can be solved by a multistage merge process, but there is a trivial solution using a min-heap data structure, where quotes are ordered by timestamp. First build the min-heap with the first quote from each file; then iteratively extract the minimum entry e from the min-heap, write it to R, and add in the next entry in the file corresponding to e. Details are given in Problem 2.10.

ITERATIVE REFINEMENT OF BRUTE-FORCE SOLUTION

Consider the problem of string search (*cf.* Problem 5.1): given two strings s (search string) and T (text), find all occurrences of s in T. Since s can occur at any offset in T, the brute-force solution is to test for a match at every offset. This algorithm is perfectly correct; its time complexity is $O(n \cdot m)$, where n and m are the lengths of s and T.

After trying some examples, you may see that there are several ways in which to improve the time complexity of the brute-force algorithm. For example, if the character $T[i]$ is not present in s you can suitably advance the matching. Furthermore, this skipping works better if we match the search string from its end and work backwards. These refinements will make the algorithm very fast (linear-time) on random text and search strings; however, the worst case complexity remains $O(n \cdot m)$.

You can make the additional observation that a partial match of s which does not result in a full match implies other offsets which cannot lead to full matches. For example, if $s = abdabcabc$ and if, starting backwards, we have a partial match up to $abcabc$ that does not result in a full match, we know that the next possible matching offset has to be at least 3 positions ahead (where we can match the second abc from the partial match).

By putting together these refinements you will have arrived at the famous Boyer-Moore string search algorithm—its worst-case time complexity is $O(n+m)$ (which is the best possible from a theoretical perspective); it is also one of the fastest string search algorithms in practice.

SMALL EXAMPLES

Problems that seem difficult to solve in the abstract, can become much more tractable when you examine small concrete instances. For instance, consider the following problem: there are 500 closed doors along a corridor, numbered from 1 to 500. A person walks through the corridor and opens each door. Another person walks through the corridor and closes every alternate door. Continuing in this manner, the i-th person comes and toggles the position of every i-th door starting from door i. You are to determine exactly how many doors are open after the 500-th person has walked through the corridor.

It is very difficult to solve this problem using abstract variables. However if you try the problem for $1, 2, 3, 4, 10$, and 20 doors, it takes under a minute to see that the doors that remain open are $1, 4, 9, 16 \ldots$, regardless of the total number of doors. The pattern is obvious—the doors that remain open are those numbered by perfect squares. Once you make this connection, it is easy to prove it for the general case. Hence the total number of open doors is $\lfloor \sqrt{500} \rfloor = 22$. Refer to Problem 9.4 for a detailed solution.

REDUCTION

Consider the problem of finding if one string is a rotation of the other, e.g., "car" and "arc" are rotations of each other A natural approach may be to rotate the first string by every possible offset and then compare it with the second string. This algorithm would have quadratic time complexity.

You may notice that this problem is quite similar to string search which can be done in linear time, albeit using a somewhat complex algorithm. So it would be natural to try to reduce this problem to string search. Indeed, if we concatenate the second string with itself and search for the first string in the resulting string, we will find a match iff the two original strings are rotations of each other. This reduction yields a linear-time algorithm for our problem; details are given in Problem 5.4.

Usually you try to reduce your problem to an easier problem. But sometimes, you need to reduce a problem known to be difficult to your given problem to show that your problem is difficult. Such problems are described in Chapter 6.

GRAPH MODELING

Drawing pictures is a great way to brainstorm for a potential solution. If the relationships in a given problem can be represented using a graph, quite often the problem can be reduced to a well-known graph problem. For example, suppose you are given a set of barter rates between commodities and you are supposed to find out if an arbitrage exists, i.e., there is a way by which you can start with a units of some commodity C and perform a series of barters which results in having more than a units of C.

We can model the problem with a graph where commodities correspond to vertices, barters correspond to edges, and the edge weight is set to the logarithm of the barter rate. If we can find a cycle in the graph with a positive weight, we would have found such a series of exchanges. Such a cycle can be solved using the Bellman-Ford algorithm (*cf*. Problem 4.19).

WRITE AN EQUATION

Some problems can be solved by expressing them in the language of mathematics. For example, suppose you were asked to write an algorithm that computed binomial coefficients, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

The problem with computing the binomial coefficient directly from the definition is that the factorial function grows very quickly and can overflow an integer variable. If we use floating point representations for numbers, we lose precision and the problem of overflow does not go away. These problems potentially exist even if the final value of $\binom{n}{k}$ is small. One can try to factor the numerator and denominator and try and cancel out common terms but factorization is itself a hard problem.

The binomial coefficients satisfy the *addition formula*:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

This identity leads to a straightforward recursion for computing $\binom{n}{k}$ which avoids the problems mentioned above. Dynamic programming has to be used to achieve good time complexity—details are in Problem 9.1.

AUXILIARY ELEMENTS

Consider an 8×8 square board in which two squares on diagonally opposite corners are removed. You are given a set of thirty-one 2×1 dominoes and are asked to cover the board with them.

After some (or a lot) of trial-and-error, you may begin to wonder if a such a configuration exists. Proving an impossibility result may seem hard. However if you think of the 8×8 square board as a chessboard, you will observe that the removed corners are of the same color. Hence the board consists of either 30 white squares and 32 black squares or vice versa. Since a domino will always cover two adjacent squares, any arrangement of dominoes must cover the same number of black and white squares. Hence no such configuration exists.

The original problem did not talk about the colors of the squares. Adding these colors to the squares makes it easy to prove impossibility, illustrating the strategy of adding auxiliary elements.

VARIATION

Suppose we were asked to design an algorithm which takes as input an undirected graph and produces as output a black or white coloring of the vertices such that for every vertex, at least half of its neighbors differ in color from it.

We could try to solve this problem by assigning arbitrary colors to vertices and then flipping colors wherever constraints are not met. However this approach does not converge on all examples.

It turns out we can define a slightly different problem whose solution will yield the coloring we are looking for. Define an edge to be *diverse* if its ends have different colors. It is easy to verify that a color assignment that maximizes the number of diverse edges also satisfies the constraint of the original problem. The number of diverse edges can be maximized greedily flipping the colors of vertices that would lead to a higher number of diverse edges; details are given in Problem 4.11.

PARALLELISM

In the context of interview questions, parallelism is useful when dealing with scale, i.e., when the problem is so large that it is impossible to solve it on a single machine or it would take a very long time. The key insight you need to display is how to decompose the problem such that (1.) each subproblem can be solved relatively independently and (2.) constructing the solution to the original problem from solutions to the subproblems is not expensive in terms of CPU time, main memory, and network usage.

Consider the problem of sorting a petascale integer array. If we know the distribution of the numbers, the best approach would be to define equal-sized ranges of integers and send one range to one machine for sorting. The sorted numbers would just need to be concatenated in the correct order. If the distribution is not known then we can send equal-sized arbitrary subsets to each machine and then merge the sorted results

using a min-heap. For details on petascale sorting, please refer to Problem 2.2.

CACHING

Caching is a great tool whenever there is a possibility of repeating computations. For example, the central idea behind dynamic programming is caching results from intermediate computations. Caching becomes extremely useful in another setting where requests come to a service in an online fashion and a small number of requests take up a significant amount of compute power. Workloads on web services exhibit this property; Problem 7.1 describes one such problem.

SYMMETRY

While symmetry is a simple concept it can be used to solve very difficult problems, sometimes in less than intuitive ways. Consider a 2-player game in which players alternately take bites from a chocolate bar. The chocolate bar is an $n \times m$ rectangle; a bite must remove a square and all squares above and to the right in the chocolate bar. The first player to eat the lower leftmost square loses (think of it as being poisoned).

Suppose we are asked whether we would prefer to play first or second. One approach is to make the observation that the game is symmetrical for Player 1 and Player 2, except for their starting state. If we assume that there is no winning strategy for Player 1, then there must be a way for Player 2 to win if Player 1 bites the top right square in his first move. Whatever move Player 2 makes after that can always be made by Player 1 as his first move. Hence Player 1 can always win. For a detailed discussion, refer to the Problem 9.13.

CONCLUSION

In addition to developing intuition for which technique may apply to which problem, it is also important to know when your technique is not working and quickly move to your next best guess. In an interview setting, even if you do not end up solving the problem entirely, you will get credit for applying these techniques in a systematic way and clearly communicating your approach to the problem. We cover nontechnical aspects of problem solving in Chapter 12.

Chapter 1

Searching

> Searching is a basic tool that every programmer should keep in mind for use in a wide variety of situations.
>
> "The Art of Computer Programming, Volume 3 - Sorting and Searching," D. Knuth, 1973

Given an arbitrary collection of n keys, the only way to determine if a search key is present is by examining each element which yields $\Theta(n)$ complexity. If the collection is "organized", searching can be sped up dramatically. Of course, inserts and deletes have to preserve the organization; there are several ways of achieving this.

Binary Search

Binary search is at the heart of more interview questions than any other single algorithm. Fundamentally, binary search is a natural divide-and-conquer strategy for searching. The idea is to eliminate half the keys from consideration by keeping the keys in a sorted array. If the search key is not equal to the middle element of the array, one of the two sets of keys to the left and to the right of the middle element can be eliminated from further consideration.

Questions based on binary search are ideal from the interviewers perspective: it is a basic technique that every reasonable candidate is supposed to know and it can be implemented in a few lines of code. On the other hand, binary search is much trickier to implement correctly than it appears—you should implement it as well as write corner case tests to ensure you understand it properly.

Many published implementations are incorrect in subtle and not-so-subtle ways—a study reported that it is correctly implemented in only five out of twenty textbooks. Jon Bentley, in his book *Programming Pearls* reported that he assigned binary search in a course for professional programmers and found that 90% percent failed to code it correctly despite having ample time. (Bentley's students would have been gratified to know that his own published implementation of binary search, in a chapter titled "Writing Correct Programs", contained a bug that remained undetected for over twenty years.)

Binary search can be written in many ways—recursive, iterative, different idioms for conditionals, etc. Here is an iterative implementation adapted from Bentley's book, which includes his bug.

```
1  public class BinSearch {
2    static int search( int [] A, int K ) {
3      int l = 0;
4      int u = A.length-1;
5      int m;
6      while ( l <= u ) {
7        m = (l+u)/2;
8        if (A[m] < K) {
9          l = m + 1;
10       } else if (A[m] == K) {
11         return m;
12       } else {
13         u = m-1;
14       }
15     }
16     return -1;
17   }
18 }
```

The error is in the assignment `m = (l+u)/2;` it can lead to overflow and should be replaced by `m = l + (u-l)/2`.

The time complexity of binary search is given by $B(n) = c + B(n/2)$. This solves to $B(n) = O(\log n)$, which is far superior to the $O(n)$ approach needed when the keys are unsorted. A disadvantage of binary search is that it requires a sorted array and sorting an array takes $O(n \log n)$ time. However if there are many searches to perform, the time taken to sort is not an issue.

We begin with a problem that on the face of it has nothing to do with binary search.

1.1 COMPUTING SQUARE ROOTS

Square root computations can be implemented using sophisticated numerical techniques involving iterative methods and logarithms. However if you were asked to implement a square root function, you would not be expected to know these techniques.

Problem 1.1: Implement a fast integer square root function that takes in a 32-bit unsigned integer and returns another 32-bit unsigned integer that is the floor of the square root of the input.

There are many variants of searching a sorted array that require a little more thinking and create opportunities for missing corner cases. For the following problems, A is a sorted array of integers.

1.2 SEARCH A SORTED ARRAY FOR k

Write a method that takes a sorted array A of integers and a key k and returns the index of first occurrence of k in A. Return -1 if k does not appear in A. Write tests to verify your code.

1.3 SEARCH A SORTED ARRAY FOR THE FIRST ELEMENT LARGER THAN k

Design an efficient algorithm that finds the index of the first occurrence an element larger than a specified key k; return -1 if every element is less than or equal to k.

1.4 SEARCH A SORTED ARRAY FOR $A[i] = i$

Suppose that in addition to being sorted, the entries of A are distinct integers. Design an efficient algorithm for finding an index i such that $A[i] = i$ or indicating that no such index exists.

1.5 SEARCH AN ARRAY OF UNKNOWN LENGTH

Suppose you do not know the length of A in advance; accessing $A[i]$ for i beyond the end of the array throws an exception.

Problem 1.5: Find the index of the first occurrence in A of a specified key k; return -1 if k does not appear in A.

1.6 MISSING ELEMENT, LIMITED RESOURCES

The storage capacity of hard drives dwarfs that of RAM. This can lead to interesting time-space tradeoffs.

Problem 1.6: Given a file containing roughly 300 million social security numbers (9-digit numbers), find a 9-digit number that is not in the file. You have unlimited drive space but only 2 megabytes of RAM at your disposal.

not change—if A beats B in one time-trial and B beats C in another time-trial, then A is guaranteed to beat C if they are in the same time-trial.

Problem 2.5: What is the minimum number of time-trials needed to determine who to send to the Olympics?

2.6 LEAST DISTANCE SORTING

You come across a collection of 20 stone statues in a line. You want to sort them by height, with the shortest statue on the left. The statues are very heavy and you want to move them the least possible distance.

Problem 2.6: Design a sorting algorithm that minimizes the total distance that the statues are moved.

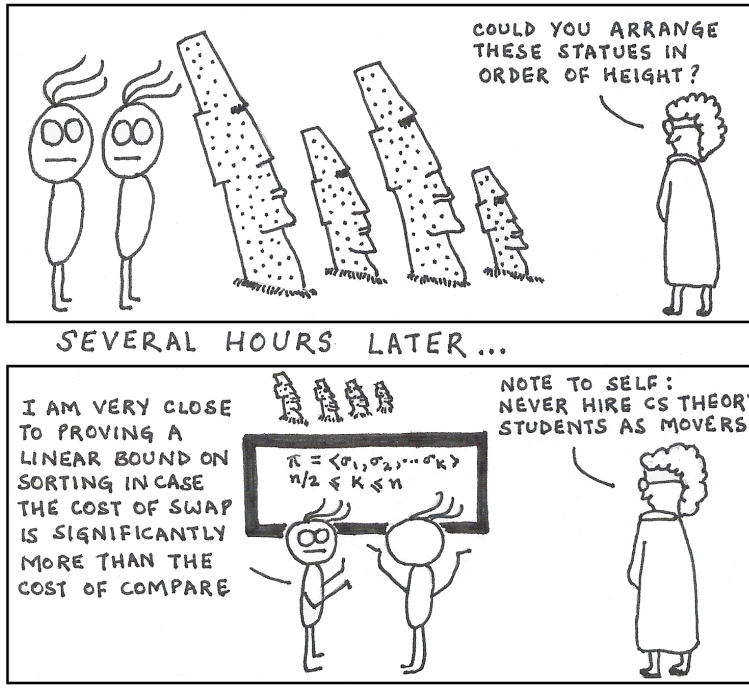

Figure 2. "Premature optimization is the root of all evil"—D. Knuth

2.7 PRIVACY AND ANONYMIZATION

The Massachusetts Group Insurance Commission had a bright idea back in the mid 1990s—it decided to release "anonymized" data on state em-

3.1 LONGEST NONDECREASING SUBSEQUENCE

In genomics, given two gene sequences, we try to find if parts of one gene are the same as the other. Thus it is important to find the longest common subsequence of the two sequences. One way to solve this problem is to construct a new sequence where for each literal in one sequence, we insert its position into the other sequence and then find the longest nondecreasing subsequence of this new subsequence. For example, if the two sequences are $\langle 1, 3, 5, 2, 7\rangle$ and $\langle 1, 2, 3, 5, 7\rangle$, we would construct a new sequence where for each position in the first sequence, we would list its position in the second sequence like so, $\langle 1, 3, 4, 2, 5\rangle$. Then we find the longest nondecreasing sequence which is $\langle 1, 3, 4, 5\rangle$. Now, if we use the numbers of the new sequence as indices into the second sequence, we get $\langle 1, 3, 5, 7\rangle$ which is our longest common subsequence.

Problem 3.1: Given an array of integers A of length n, find the longest sequence $\langle i_1, \ldots i_k\rangle$ such that $i_j < i_{j+1}$ and $A[i_j] \leq A[i_{j+1}]$ for any $j \in [1, k-1]$.

3.2 FROG CROSSING

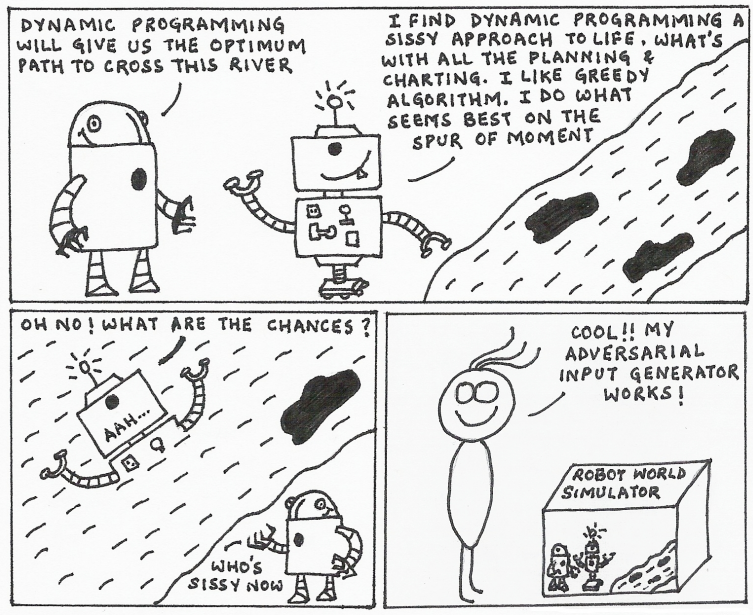

Figure 3. "Be fearful when others are greedy"—W. Buffett

DP is often used to compute a plan for performing a task that consists of a series of actions in an optimum way. Here is an example with an interesting twist.

Problem 3.2: There is a river that is n meters wide. At every meter from the edge, there may or may not be a stone. A frog needs to cross the river. However the frog has the limitation that if it has just jumped x meters, then its next jump must be between $x - 1$ and $x + 1$ meters, inclusive. Assume the first jump can be of only 1 meter. Given the position of the stones, how would you determine whether the frog can make it to the other end or not? Analyze the runtime of your algorithm.

3.3 CUTTING PAPER

We now consider an optimum planning problem in two dimensions. You are given an $L \times W$ rectangular piece of kite-paper, where L and W are positive integers and a list of n kinds of kites that can be made using the paper. The i-th kite design, $i \in [1, n]$ requires an $l_i \times w_i$ rectangle of kite-paper; this kite sells for p_i. Assume l_i, w_i, p_i are positive integers. You have a machine that can cut rectangular pieces of kite-paper either horizontally or vertically.

Problem 3.3: Design an algorithm that computes a profit maximizing strategy for cutting the kite-paper. You can make as many instances of a given kite as you want. There is no cost to cutting kite-paper.

3.4 WORD BREAKING

Suppose you are designing a search engine. In addition to getting keywords from a page's content, you would like to get keywords from URLs. For example, `bedbathandbeyond.com` should be associated with "bed bath and beyond" (in this version of the problem we also allow "bed bat hand beyond" to be associated with it).

Problem 3.4: Given a dictionary that can tell you whether a string is a valid word or not in constant time and given a string s of length n, provide an efficient algorithm that can tell whether s can be reconstituted as a sequence of valid words. In the event that the string is valid, your algorithm should output the corresponding sequence of words.

The next three problems have a very similar structure. Given a set of objects of different sizes, you need to partition them in various ways. The solutions also have the same common theme that you need to explore all possible partitions in a way that you can take advantage of overlapping subproblems.

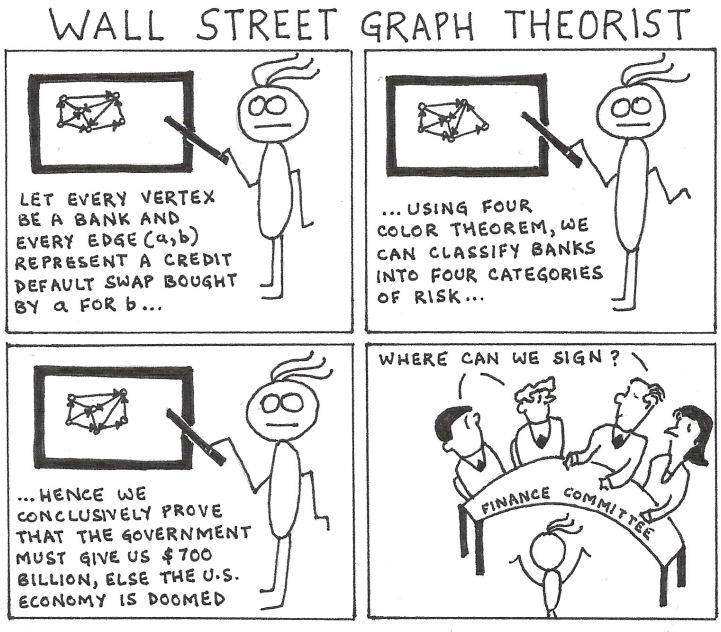

Figure 4. The power of obscure proofs

is called the root, an ordered tree is a rooted tree in which each vertex has an ordering on its children, etc.

Graph Search

Computing vertices which are reachable from other vertices is a fundamental operation. There are two basic algorithms—Depth First Search (DFS) and Breadth First Search (BFS). Both are linear-time—$O(|V| + |E|)$. They differ from each other in terms of the additional information they provide, e.g., BFS can be used to compute distances from the start vertex and DFS can be used to check for the presence of cycles.

4.1 Searching a maze

It is natural to apply graph models and algorithms to spatial problems. Consider a black and white digitized image of a maze—white pixels represent open areas and black spaces are walls. There are two special pixels: one is designated the entrance and the other is the exit.

Problem 5.5: Implement a function which takes a URL as input and performs the following transformations on it: (1.) make hostname and protocol lowercase, (2.) if it ends in index.html or default.html, remove the filename, (3.) if protocol field is missing, add "http://" at the beginning, and (4.) replace consecutive '/' characters by a single '/' in the "path" segment of the URL.

5.6 LONGEST PALINDROME SUBSEQUENCE

A palindrome is a string which is equal to itself when reversed. For example, the human Y-chromosome contains a gene with the amino acid sequence $\langle C, A, C, A, A, T, T, C, C, C, A, T, G, G, G, T, T, G, T, G, G, A, G \rangle$, which includes the palindromic subsequences $\langle T, G, G, G, T \rangle$ and $\langle T, G, T \rangle$. Palindromic subsequences in DNA are significant because they influence the ability of the strand to loop back on itself.

Problem 5.6: Devise an efficient algorithm that takes a DNA sequence $D[1, \ldots, n]$ and returns the length of the longest palindromic subsequence.

5.7 PRETTY PRINTING

Consider the problem of arranging a piece of text in a fixed width font (i.e., each character has the same width) in a rectangular space. Breaking words across line boundaries is visually displeasing. If we avoid word breaking, then we may frequently be left with many spaces at the end of lines (since the next word will not fit in the remaining space). However if we are clever about where we break the lines, we can reduce this effect.

Problem 5.7: Given a long piece of text, decompose it into lines such that no word spans across two lines and the total wasted space at the end of each line is minimized.

5.8 EDIT DISTANCES

Spell checkers make suggestions for misspelled words. Given a misspelled string s, a spell checker should return words in the dictionary which are close to s.

One definition of closeness is the number of "edits" it would take to transform the misspelled word into a correct word, where a single edit is the deletion or insertion of a single character.

Problem 5.8: Given two strings A and B, compute the minimum number of edits needed to transform A into B.

6.4 COMPUTING x^n

A straight-line program for computing x^n is a finite sequence

$$x \mapsto x^{i_1} \mapsto x^{i_2} \mapsto \cdots \mapsto x^n$$

constructed as follows: the first element is x; each succeeding element is either the square of some previously computed element or the product of any two previously computed elements. The number of multiplications to evaluate x^n is the number of terms in the shortest such program sequence minus one. No efficient method is known for the problem of determining the minimum number of multiplications needed to evaluate x^n; the problem for multiple exponents is known to be NP-complete.

Problem 6.4: How would you determine the minimum number of multiplications to evaluate x^{30}?

6.5 CNF-SAT

The CNF-SAT problem was defined in Problem 4.24. In that problem, we asked for a linear-time algorithm for the special case where each clause had exactly two literals.

Problem 6.5: Design an algorithm for CNF-SAT. Your algorithm should use branch-and-bound to prune partial assignments that can easily be shown to be unsatisfiable.

The following problems illustrate the use of heuristic search and pruning principles.

6.6 SCHEDULING

We need to schedule N lectures in M classrooms. Some of those lectures are prerequisites for others.

Problem 6.6: How would you choose when and where to hold the lectures in order to finish all the lectures as soon as possible?

6.7 HARDY-RAMANUJAN NUMBER

The mathematician G. H. Hardy was on his way to visit his collaborator S. Ramanujan who was in the hospital. Hardy remarked to Ramanujan that he traveled in taxi cab number 1729 which seemed a dull one and he hoped it was not a bad omen. To this, Ramanujan replied that 1729 was a very interesting number—it was the smallest number expressible as the sum of cubes of two numbers in two different ways. Indeed, $10^3 + 9^3 = 12^3 + 1^3 = 1729$.

7.2 THREAD POOLS

The following class, `SimpleWebServer`, implements part of a simple HTTP server:

```
1  public class SimpleWebServer {
2    final static int PORT = 8080;
3    public static void main (String [] args) throws IOException
         {
4      ServerSocket serversock = new ServerSocket(PORT);
5      for (;;) {
6        Socket sock = serversock.accept();
7        ProcessReq(sock);
8      }
9    }
10 }
```

Problem 7.2: Suppose you find that `SimpleWebServer` has poor performance because `processReq` frequently blocks on IO. What steps could you take to improve `SimpleWebServer`'s performance?

7.3 ASYNCHRONOUS CALLBACKS

It is common in a distributed computing environment for the responses to not return in the same order as the requests were made. One way to handle this is through an "asynchronous callback"—a method to be invoked on response.

Problem 7.3: Implement a `Requestor` class. The class has to implement a `Dispatch` method which takes a `Requestor` object. The `Requestor` object includes a `request` string, a `ProcessResponse(string response)` method, and an `Execute` method that takes a string and returns a string.

`Dispatch` is to create a new thread which invokes `Execute` on `request`. When `Execute` returns, `Dispatch` invokes the `ProcessResponse` method on the response.

The `Execute` method may take an indeterminate amount of time to return; it may never return. You need to have a time-out mechanism for this: assume the `Requestor` objects have an `Error` method that you can invoke.

7.4 TIMER

Consider a web-based calendar in which the server hosting the calendar has to perform a task when the next calendar event takes place. (The task could be sending an email or an SMS.) Your job is to design a facility that manages the execution of such tasks.

Chapter 8

Design Problems

> We have described a simple but very powerful and flexible protocol which provides for variation in individual network packet sizes, transmission failures, sequencing, flow control, and the creation and destruction of process-to-process associations.
>
> "A Protocol for Packet Network Intercommunication," V. Cerf and R. Kahn, 1974

This chapter is concerned with system design problems. Each question can be a large open-ended software project. During the interview, you should provide a high level sketch of such a system with thoughts on various design choices, the tradeoffs, key algorithms, and the data-structures involved.

8.1 MOSAIC

One popular form of computer art is photomosaics where you are given a collection of images called "tiles". Then given a target image, you want to build another image which closely approximates the target image but is actually built by juxtaposing the tiles. Here the quality of approximation is mostly defined by human perception. It is often the case that with a given set of tiles, a user may want to build several mosaics.

Problem 8.1: How would you design a software that produces high quality mosaics with minimal compute time?

is dropped from any floor X or higher but will remain intact if dropped from a floor below X.

Problem 9.5: Given K balls and N floors, what is the minimum number of ball drops that are required to determine X in the worst-case?

9.6 BETTING ON CARD COLORS

A deck of 52 playing cards is shuffled. The deck is placed face-down on a table. You can place a bet on the color of the top card at even odds. After you have placed your bet, the card is revealed to you and discarded. Betting continues till the deck is exhausted. On any card, you can bet any amount from 0 to all the money you have and the odds are always even.

Problem 9.6: You begin with one dollar. It is known that if you can bet arbitrary fractions of the money you have, the maximum amount of money that you guarantee you can win, regardless of the order in which the cards appear, is $2^{52}/\binom{52}{26} \approx 9.08132955$. However you are allowed to bet only in penny increments. Write a program to compute a tight lower bound on the amount you can win under this restriction.

Invariants

The following problem was popular at interviews in the early 1990s: you are given a chessboard with two squares at the opposite ends of a diagonal removed, leaving 62 squares. You are given 31 rectangular dominoes. Each can cover exactly two squares. How would you cover all the 62 squares with the dominoes?

It is easy to spend hours trying unsuccessfully to find such a covering. This will teach you that a problem may be intentionally worded to mislead you into following a futile path.

There is a simple argument that no covering exists—the two squares removed will always have the same color, so there will be either 30 black and 32 white squares to be covered or 32 black and 30 white squares to be covered. Each domino will cover one black and one white square, so the number of black and white squares covered as you successively put down the dominoes is equal. Hence it is impossible to cover the given chessboard.

This proof of impossibility is an example of invariant analysis. An invariant is a function of the state of a system being analyzed that remains constant in the presence of (possibly restricted) updates to the state. Invariant analysis is particularly powerful at proving impossibility results as we just saw with the chessboard tiling problem. The challenge is finding a simple invariant.

of the inter-arrival time of requests. Now, in your load test you want to generate requests for your server such that the inter-arrival times come from the same distribution that you have seen in your data. How would you generate these inter-arrival times?

Problem 10.10: Given the probability distribution of a discrete random variable X and a uniform $[0, 1]$ random number generator, how would you generate instances of X that follow the given distribution?

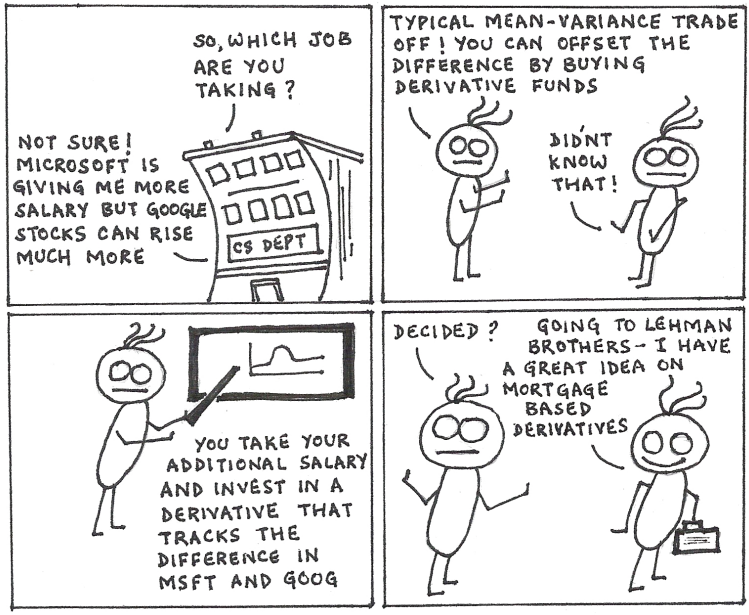

Figure 6. *FINANCIAL ENGINEERING*: an oxymoron widely used *circa* 2008.

10.11 EXPECTED NUMBER OF DICE ROLLS

Bob repeatedly rolls an unbiased 6-sided dice. He stops when he has rolled all the six numbers on the dice. How many rolls will it take, on an average, for Bob to see all the six numbers?

Option pricing

A call option gives the owner the right to buy something—a share, a barrel of oil, an ounce of gold—at a predetermined price at a predetermined time in the future. If the option is not priced fairly, an arbitrageur

can either buy or sell the option in conjunction with other transactions and come up with a scheme of making money in a guaranteed fashion. A fair price for an option would be a price such that no arbitrage scheme can be designed around it.

We now consider problems related to determining the fair price for an option for a stock, given the distribution of the stock price for a period of time.

10.12 OPTION PRICING—DISCRETE CASE

Consider an option to buy a stock S that currently trades at \$100. The option is to buy the stock at \$100 in 100 days.

Suppose we know there are only two possible outcomes—S will go to \$120 or to \$70.

An arbitrage is a situation where you can start with a portfolio (x_s shares and x_o options) which has negative value (since you are allowed to short shares and sell options, both x_s and x_o may be negative) and regardless of the movement in the share price, the portfolio has positive value.

For example, if the option is priced at \$26, we can make money by buying one share for \$100 and selling four options—the initial outlay on the portfolio is $100 - 4 \times 26 = -4$. If the stock goes up, our portfolio is worth $120 - 20 \times -4 = \$80$. If the stock goes down, the portfolio is worth $\$70$. In either case, we make money with no initial investment, i.e., the option price allows for an arbitrage.

Problem 10.12: For what option price(s), are there no opportunities for arbitrage?

10.13 OPTION PRICING WITH INTEREST

Consider the same problem as Problem 10.12, with the existence of a third asset class, a bond. A \$1 bond pays \$1.02 in 100 days. You can borrow money at this rate or lend it at this rate.

Problem 10.13: Show there is a unique arbitrage-free price for the option and compute this price.

10.14 OPTION PRICING—CONTINUOUS CASE

Problem 10.14: Suppose the price of Jingle stock 100 days in the future is a normal random variable with mean \$300 and deviation \$20. What would be the fair price of an option to buy a single share of Jingle at \$300 in 100 days worth today? (Ignore the impact of interest rates.)

11.13 BEST PRACTICES

Our favorite best practices book is *Effective Java* by Bloch—it covers many topics: object-oriented programming, design patterns, code organization, concurrency, and generics are just a few examples. *Effective C++* by Meyer is highly thought of for C++. *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma *et al.* is a very popular introduction to patterns.

- Give an example of a problem you solved where you made good use of object-oriented programming.
- What is the factory pattern? What is the publish-subscribe model?
- Give an example of how inheritance violates encapsulation.
- What do Java bounded wildcards buy you?
- Why should you always override the equals and hash function methods for Java classes?

11.14 PROGRAMMING LANGUAGE IMPLEMENTATION

We recommend *Programming Languages Pragmatics* by Scott—it covers both theory and practice of programming languages and is very up-to-date.

- Give an example of a language which cannot be parsed by any computer.
- What problems does dynamic linkage solve? What problems does it introduce?
- What is a functional language?
- What is a virtual function?
- How is method dispatch implemented in Java?
- What is automatic garbage collection and how is it implemented?
- What is a type-safe language?
- What is the difference between a lexer and a parser?
- Give an example of a language which cannot be recognized by a lexer.
- Give an example of a language which cannot be parsed by a parser.

11.15 OPERATING SYSTEMS

Modern Operating Systems by Tanenbaum is widely used; one of its claims to fame is that Linux was developed from the Minix OS developed in an earlier version of this book.

- What is a system call?
- How is a system call different from a library call?

Chapter 12

Strategies For A Great Interview

A typical one hour interview with a single interviewer consists of five minutes of introductions and questions about the candidate's resumé. This is followed by five to fifteen minutes of questioning on basic programming concepts.

The core of the interview is one or two detailed algorithm design questions where the candidate is expected to present a detailed solution on a whiteboard or paper. Depending on the interviewer and the question, the solution may be required to include syntactically correct code in a language that the candidate is comfortable with.

The reason for asking such questions is that algorithms and associated data-structures underlie all software. They are often hidden in library calls. They can be a small part of a code base dominated by IO and format conversion. But they are the crucial component in terms of performance and intricacy.

The most important part of interview preparation is to know the material and practice solving problems. However the nontechnical aspects of interviewing cannot be underplayed either. There are a number of things that could go wrong in an interview and it is important to have a strategy to deal with them.

12.1 Before the interview

One of the best ways of preparing for an interview is mock interviews. Get a friend to ask you questions from this book (or any other source) and have you solve the problems on a whiteboard or paper. Ask your friend to take notes and give you detailed feedback, both positive and negative. Also ask your friend to provide hints from the solution if you are stuck. This will help you overcome any fear or problem areas well in advance.

Chapter 1

Searching

Solution 1.1: One of the fastest ways to invert a fast-growing monotone function (such as the square function) is to do a binary search in a precomputed table of the function. Since the square root for the largest 32-bit unsigned integer can be represented in 16 bits, we build an array of length 2^{16} such that i-th element in the array is i^2. When we want to compute square root for a given number n, we look for the largest number in the array that is still smaller than n. Because the square root is relatively small, it is faster to compute it on the fly than to precompute it.

```
1  unsigned int sqrt_search(unsigned int input) {
2    int begin = 0;
3    int end = 65536;
4    while(begin + 1 < end){
5      int mid = begin + (end - begin) / 2;
6        unsigned int mid_sqr = mid * mid;
7      if (mid_sqr == input) {
8        return mid;
9      } else if (mid_sqr > input) {
10       end = mid;
11     } else {
12       begin = mid;
13     }
14   }
15   return begin;
16 }
```

Solution 1.2:

```
1  public class BinSearch {
2    static int search( int [] A, int K ) {
3      int l = 0;
4      int u = A.length-1;
5      int m;
6      while ( l <= u ) {
```

```
7         m = (l+u)/2;
8         if (A[m] < K) {
9           l = m + 1;
10        } else if (A[m] == K) {
11          return m;
12        } else {
13          u = m-1;
14        }
15      }
16      return -1;
17    }
18  }
```

Solution 1.3: A straightforward way to find an element larger than a given value k is to look for k via a binary search and then, if k is found, walk the array forward (linearly) until either the first element larger than k is encountered or the end of the array is reached. If k is not found, a binary search will end up pointing to either the next largest value after K in the array, in which case no further action is required or the next smallest value in which case the next element is the next largest value.

The worst-case runtime of this algorithm is $\Theta(n)$—the input of all values matching K, except for the last one (which is greater than K), is the worst-case.

The solution to this problem is to replace the linear scan with a binary search in the second part of the algorithm, which leads to the desired element to be found in $O(\log n)$ time.

Solution 1.4: Since the array contains distinct integers and is sorted, for any $i > 0$, $A[i] \geq A[i-1] + 1$. Therefore $B[i] = A[i] - i$ is also nondecreasing. It follows that we can do a binary search for 0 in B to find an index such that $A[i] = i$. (We do not need to actually create B, we can simply use $A[i] - i$ wherever $B[i]$ is referenced.)

Solution 1.5: The key idea here is to simultaneously do a binary search for the end of the array as well as the key. We try to look for $A[2^k]$ in the k-th step and catch exceptions for successive values of k till either we hit an exception or we hit a number greater than or equal to b. Then we do a binary search for b between indices 2^{k-1} and 2^k. The runtime of the search algorithm is $O(\log n)$. In code:

```
1   int BinarySearchInUnboundedArray(int * A, int b) {
2     int k = 0;
3     while(true) {
4       int c;
5       try {
6         c = A[(1 << k) -1];
7         if (c == b) {
8           return (1 << k) -1;
```

we can write the following recurrence relationship:

$$A(p_1, \ldots, p_k) = \min_{x:3 \le x \le k} \Big(A(p_3, \ldots, p_x) + A(p_x, \ldots, p_k, p_1) + L(p_1, p_2) + L(p_1, p_x) + L(p_2, p_x) \Big).$$

If we tabulate the cost of triangulation of each polygon that is a result of picking subsequent points on the original polygon, we would need to do this for roughly n^2 polygons. If we have already tabulated the value for all smaller polygons, it will take us $O(n)$ time for doing so. Hence we can compute the minimum cost in $O(n^3)$ time.

Solution 3.11: We focus on the case where all the operands are nonnegative integers and the only operations are $\cdot$ and $+$.

Represent the expression $v_0 \circ_0 v_1 \circ_1 \cdots \circ_{n-2} v_{n-1}$ by arrays $V = [v_0, \ldots, v_{n-1}]$ and $\circ_0, \ldots, \circ_{n-2}$.

Let Max$[i, j]$ denote the maximum value achievable by some parenthesization for the subexpression $v_i \circ_i v_i \circ_i \cdots \circ_j v_j$, where Max$[i, j]$ is just $V[i]$.

The key to solving this problem is to recognize that if operation $\circ_i$ is performed last, the subexpressions $v_0 \circ_0 v_1 \circ_1 \cdots \circ_{i-2} v_{i-1}$ and $v_{i+1} \circ_{i+1} \cdots \circ_{n-2} v_{n-1}$ must be parenthesized to be maximized individually.

In particular, the maximum value must be achieved for some value of i in $[0, n-2]$, so

$$\text{Max}[0, n-1] = \max_{i \in [0, n-2]} \text{Max}[0, i] \circ_i \text{Max}[i+1, n-1].$$

The total number of recursive calls is $O(\binom{n}{2})$ and each call requires $O(n)$ additional computation to combine the results, leading to an $O(n^3)$ algorithm.

Efficiently computing this recurrence requires that intermediate results be cached. In code:

```
1  public class Parens {
2
3    int [] V;
4    char [] Op;
5
6    int [][] Max;
7    boolean [][] valid;
8
9    public Parens(int [] V, char [] Op) {
10     this.V = V;
11     this.Op = Op;
12   }
13
```

```
14    public int maxExpr(int begin, int end) {
15
16      if ( valid[begin][end] ) {
17        return Max[begin][end];
18      }
19
20      if ( begin == end ) {
21        Max[begin][end] = V[begin];
22        valid[begin][end] = true;
23        return V[begin];
24      }
25
26      if ( begin + 1 == end ) {
27        Max[begin][end] = (Op[begin] == '+') ?
28                            V[begin] + V[end] :
29                            V[begin] * V[end];
30        valid[begin][end] = true;
31        return Max[begin][end];
32      }
33
34      int max = Integer.MIN_VALUE;
35      int candidateMax = 0;
36      for ( int i = begin + 1; i < end; i++ ) {
37        int lMax = maxExpr( begin, i);
38        int rMax = maxExpr( i+1, end);
39        if ( Op[i] == '+' ) {
40          candidateMax = lMax + rMax;
41        } else {
42          candidateMax = lMax * rMax;
43        }
44        max = (max < candidateMax) ? candidateMax : max;
45      }
46      Max[begin][end] = max;
47      valid[begin][end] = true;
48      return max;
49    }
50
51    public int maxExpr() {
52      int N = V.length;
53      Max = new int [N][N];
54      valid = new boolean [N][N];
55      return  maxExpr(0,N-1);
56    }
57
58    public static void main( String[] args ) {
59
60      int [] v1 = {1,2,3,3,2,1};
61      char [] o1 = {'+','*','*','+', '+'};
62
63      Parens exp1 = new Parens(v1, o1);
64      System.out.println("Max value of expression is:" + exp1.
            maxExpr() );
65    }
66  }
```

The problem with this approach is that we do not control the number of threads launched. A thread consumes a nontrivial amount of resources by itself—there is the overhead of starting and ending down the thread and the resources used by the thread. For a lightly-loaded server, this may not be an issue but under load, it can result in exceptions that are challenging, if not impossible, to handle.

The right trade-off is to use a *thread pool*. As the name implies, this is a collection of threads, the size of which is bounded. Java provides thread pools through the `Executor` framework.

```
1  class TaskExecutionWebServer {
2    private static final int NTHREADS = 100;
3    private static final Executor exec
4        = Executors.newFixedThreadPool(NTHREADS);
5
6    public static void main(String[] args) throws IOException {
7      ServerSocket socket = new ServerSocket(80);
8      while (true) {
9        final Socket connection = socket.accept();
10       Runnable task = new Runnable() {
11         public void run() {
12           handleRequest(connection);
13         }
14       };
15       exec.execute(task);
16     }
17   }
18 }
```

Solution 7.3: Our strategy is to launch a thread T per `Requestor` object. Thread T in turn launches another thread, S, which calls `execute` and `ProcessResponse`. The call to `execute` in S is wrapped in a try-catch `InterruptedException` loop; if `execute` completes successfully, `ProcessResponse` is called on the result.

After launching S, T sleeps for the timeout interval—when it wakes up, it interrupts S. If S has completed, nothing happens; otherwise, the try-catch `InterruptedException` calls `error`.

Code for this is given below:

```
1  class Requestor {
2    public String execute(String req) {
3      return "response:" + req;
4    }
5    public String error(String req) {
6      return "response:" + req + ":" + "TIMEDOUT";
7    }
8    public String execute(String req, long delay) {
9      try {
10       Thread.sleep(delay);
11     } catch (InterruptedException e) {
```

which means that the probability of H is given by

$$\frac{k}{n+1} \cdot \frac{1}{k} \cdot (n+1-k) \cdot \frac{1}{\binom{n}{k}} = \frac{(n+1-k)(n-k)!k!}{(n+1)n!} = \binom{n+1}{k},$$

so induction goes through for subsets including the $n+1$-th element.

Solution 10.3: We can make use of the algorithm for problem 10.1 with the array A initialized by $A[i] = i$. We do not actually need to store the elements in A, all we need to do is store the elements as we select them, so the storage requirement is met.

Solution 10.4: The process does not yield all permutations with equal probability. One way to see this is to consider the case $n = 3$. There are $3! = 6$ permutations possible. There are a total of $3^3 = 27$ ways in which we can choose the elements to swap and they are all equally likely. Since 27 is not divisible by 6, some permutations correspond to more ways than others, ergo not all permutations are equally likely.

The process can be fixed by selecting elements at random and moving them to the end, similar to how we proceeded in Problems 10.1 and 10.3.

Solution 10.5: Our solution to Problem 10.1 can be used with $k = n$. Although the subset that is returned is unique (it will be $\{0, 1, \ldots, n-1\}$), all $n!$ possible orderings of the elements in the set occur with equal probability. (Note that we cannot use the trick to reduce the number of calls to the random number generator at the end of Solution 10.1.)

Solution 10.6: The first thing to note is that three segments can make a triangle iff no one segment is longer than the sum of the other two: the "only if" follows from the triangle inequality and the "if" follows from a construction—take a segment and draw circles at the endpoints with radius equal to the lengths of the other circles.

Since the three segment lengths add up to 1, there is a segment that is longer than the sum of the other two iff there is a segment that is longer than $\frac{1}{2}$.

Let $l = \min(u1, u2)$, $m = \max(u1, u2) - \min(u1, u2)$, and $u = 1 - \max(u1, u2)$; these are the lengths of the first, second, and third segments, from left to right. If one segment is longer than 0.5, then none of the others can be longer than 0.5; so, the events $l > 0.5$, $m > 0.5$, and $u > 0.5$ are disjoint.

Observe that $l > 0.5$ iff both $u1$ and $u2$ are greater than 0.5; the probability of this event is $\frac{1}{2} \times \frac{1}{2}$ because $u1$ and $u2$ are chosen independently. Similarly $m > 0.5$ iff both $u1$ and $u2$ are less than 0.5, which is $\frac{1}{2} \times \frac{1}{2}$.

To compute the probability of $m > 0.5$, first we consider the case that $u1 < u2$. For $m > 0.5$, we need $u1$ to be between 0 and 1 and $u2$ to be

Index of Problems